UPC Language Specifications
V1.1

Tarek A. El-Ghazawi
George Washington University
tarek@gwu.edu

William W. Carlson   Jesse M. Draper
IDA Center for Computing Sciences
wwc@super.org   jdraper@super.org

March 24, 2003

Acknowledgments

Contents

Introduction

1   UPC is a parallel extension to ANSI C. UPC follows the distributed shared-memory programming paradigm. The first version of UPC, known as version 0.9, was published in May of 1999 as technical report [CARLSON99] at the Institute for Defense Analyses Center for Computing Sciences.

2   Version 1.0 of UPC has been initially discussed at the UPC workshop, held in Bowie, Maryland, 18-19 May, 2000. The workshop had about 50 participants from industry, government, and academia. This version was adopted with modifications in the UPC mini workshop meeting held during Supercomputing 2000, in November 2000, in Dallas, and finalized in February 2001.

3   Version 1.1 of UPC was initially discussed at the UPC workshop, held in Washington, DC, 3-5 March, 2002, and finalized in March 2003.

1   Scope

1   This document focuses only on the UPC specifications that extend ANSI C to an explicit parallel C based on the distributed shared memory model. All ANSI C specifications as per ISO/SEC 9899 [ISO/SEC00] are considered a part of these UPC specifications, and therefore will not be addressed in this document.

2   Small parts of ANSI C [ISO/SEC00] may be repeated for self-containment and clarity of a subsequent UPC extension definition.

2   Normative references

1   The following document and its identified normative references in addition to these documents constitute provisions of these UPC specifications. This will not apply to subsequent revisions of this document.

2   ISO/SEC 9899: 1999(E), Programming languages - C [ISO/SEC00]

3   The section numbering of this document is consistent with the previous document [ISO/SEC00]. The correspondence between the subsection of this document and the previous document, however, is given in Appendix A.

4   In the beginning of each UPC specifications subsection, the corresponding ANSI-C [ISO/SEC00] subsection will be noted.

3   Terms, definitions and symbols

1   For the purpose of these specifications the following definitions apply.

2   Other terms are defined where they appear in *italic* type or on the left hand side of a syntactical rule.

3.1

1   **access**
<execution-time action> to read or modify the value of an object by a thread.

3.1.1

1   **local access**
<execution-time action> to read or modify, by a given thread, the value of an object in either the private space of that thread, or in the shared address locations that have affinity to that thread.

3.1.2

1   **private access**
<execution-time action> to read or modify, by a given thread, the value of an object in the private address space of that thread.

3.1.3

1   **remote access**
<execution-time action> to read or modify, by a given thread, the value of an object whose address is in the shared address space portion which has affinity to the other threads.

3.2

1   **affinity**
a logical association of a portion of the shared address space with a given thread.

3.3

1   **shared object**
an object that resides in the shared address space.

3.4

1 **pointer-to-shared**

a pointer to a shared object.

3.5

1 **thread**

a program task in execution with access not only to a private memory space, but also to a shared memory space which can be accessed by other threads.

3.6

1 **collective**

a requirement placed on some language operations which constrains invocations of such operations to be matched[1] across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.

3.7

1 **single-valued**

an operand to a collective operation, which has the same value on every thread. The behavior of the operation is otherwise undefined.

4   Conformance

1   In this document, "shall" is to be interpreted as a requirement on a UPC implementation; conversely, "shall not" is to be interpreted as a prohibition.

2   If a "shall" or "shall not" requirement of a constraint is violated, the behavior will be undefined. Undefined behavior is indicated by "undefined behavior" or by the omission of any explicit definition of behavior from the UPC specification.

---

[1]A collective operation need not provide any actual synchronization between threads, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely upon such unspecified synchronization for correctness.

5 Environment

5.1 Conceptual Models

5.1.1 Translation environment

5.1.1.1 Threads environment

1   A UPC program is translated under either a "static THREADS" environment or a "dynamic THREADS" environment. Under the static THREADS environment, the number of threads to be used in execution is indicated to the translator in an implementation-defined manner. If the actual execution environment differs from this number of threads, the behavior of the program is undefined.

5.1.2 Execution environment

1   This subsection provides the UPC parallel extensions of [ISO/SEC00: Sec. 5.1.2]

2   Each thread has local data on which it operates and which are logically divided into a private portion and a shared portion. All operations on the private portion of the data are exactly as described in [ISO/SEC00].

3   Each thread may access shared data that have affinity to any thread; the semantics of these accesses are described herein.

4   Except for barriers automatically inserted at thread startup and termination, there is no implicit synchronization among the threads.

5   Some library calls may imply synchronization among threads. These will be explicitly noted.

5.1.2.1 Program startup

1   In the execution environment of a UPC program, derived from the hosted environment as defined in ANSI C [ISO/SEC00], each thread calls the UPC program's `main()` function[2].

---

[2]e.g., in the program `main(){ printf("hello"); }` , each thread prints `hello`.

5.1.2.2 Program termination

1   A program is terminated by the termination of all the threads[3] or a call to the function `upc_global_exit()`.

2   Thread termination follows the ANSI C definition of program termination in [ISO/SEC00: Sec. 5.1.2.2.3]. A thread is terminated by reaching the `}` that terminates the main function, by a call to the exit function, or by a return from the initial main. Note that thread termination does not imply the completion of all I/O and that shared data allocated by a thread either statically or dynamically shall not be freed before UPC program termination.

Forward references: `upc_global_exit` (7.2).

5.1.2.3 Program execution

1   References to shared objects shall be either strict or relaxed. For relaxed references there is no change to the ANSI C execution model as applied to an individual thread. This implies that translators are free to reorder and/or ignore operations (including shared operations) as long as the restrictions found in [ISO/SEC00: Sec. 5.1.2.3] are observed.

2   A further restriction applies to strict references. For each strict reference, the restrictions found in [ISO/SEC00: Sec. 5.1.2.3] must be observed with respect to all threads if that reference is eliminated (or reordered with respect to all other shared references in its thread).

3   Equally, the behavior of strict shared references can be defined as follows. Label each shared access S(i,j) or R(i,j), where S represents a strict shared access (read or write), R represents a relaxed shared access (read or write), i is the thread number making the access, j is an integer which monotonically increases as the evaluation of the program (in the abstract machine) proceeds from startup through termination. The "abstract order" is a partial ordering of all accesses by all threads such that an access x(a,b) occurs before y(c,d) in the ordering if a==c and b < d. The "actual order(k)" for thread k is another partial order in which x(a,b) occurs before y(c,d) if thread k observes the x access before it observes the y access. A thread observes all accesses present in the abstract order which affect either the data written to files by it

---

[3]A barrier is automatically inserted at thread termination.

8

or its input and output dynamics as described in [ISO/SEC00: Sect 5.1.2.3]. The least requirements on a conforming implementation are that:

- x(a,b) must "occur before" y(c,d) in actual order(e) if a == c and a == e and b < d

- x(a,b) must "occur before" y(c,d) in actual order(e) if a == c and b < d and ((x == S) or (y == S)), for all e

UNLESS such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements described in [ISO/SEC00: Sec. 5.1.2.3].

6 Language

6.1 Notations

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that

$$\{ \ expression_{opt} \ \}$$

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

6.2 Predefined identifiers

1 This subsection provides the UPC parallel extensions of section 6.4.2.2 in [ISO/SEC00].

6.2.1 THREADS

1 `THREADS` is a value of type int; it specifies the number of independent computational units and has the same value on every thread. Under the static THREADS translation environment, `THREADS` is an integer constant suitable for use in `#if` preprocessing directives.

6.2.2 MYTHREAD

1 `MYTHREAD` is a value of type int; it specifies the unique thread index. The range of possible values is `0..THREADS-1`[4].

6.2.3 UPC_MAX_BLOCK_SIZE

1 `UPC_MAX_BLOCK_SIZE` is a predefined integer constant value. It indicates the maximum value[5] allowed in a layout qualifier for shared data. It shall be suitable for use in `#if` preprocessing directives.

6.3   Expressions

1 This subsection provides the UPC parallel extensions of section 6.5 in [ISO/SEC00].

6.3.1   The `upc_localsizeof` operator

> `upc_localsizeof` *unary-expression*
>
> `upc_localsizeof` (*type-name*)

Constraints

1 The `upc_localsizeof` operator shall apply only to shared objects or shared-qualified types. All constraints on the `sizeof` operator [ISO/SEC00 Section 6.5.3.4] also apply to this operator.

Semantics

---

[4]e.g., the program `main(){ printf("%d ",MYTHREAD); }` , prints the numbers 0 thru THREADS-1, in some order.

[5]e.g. `shared [UPC_MAX_BLOCK_SIZE+1] char x[UPC_MAX_BLOCK_SIZE+1]` and `shared [*] char x[(UPC_MAX_BLOCK_SIZE+1)*THREADS]` are compile errors.

1   The `upc_localsizeof` operator returns the size, in bytes, of the local portion of its operand, which may be a shared object or a shared-qualified type. It returns the same value on all threads; the value is the maximum of the size allocated to objects with affinity to any single thread.

2   The type of the result is `size_t`.

6.3.2   The `upc_blocksizeof` operator

>   upc_blocksizeof *unary-expression*
>
>   upc_blocksizeof (*type-name*)

Constraints

1   The `upc_blocksizeof` operator shall apply only to shared objects or shared-qualified types. All constraints on the `sizeof` operator [ISO/SEC00 Section 6.5.3.4] also apply to this operator.

Semantics

1   The `upc_blocksizeof` operator returns the block size of the operand, which may be a shared object or a shared-qualified type. The block size is the value specified in the layout qualifier of the type declaration. If there is no layout qualifier, the block size is 1. The result of `upc_blocksizeof` is a compile-time constant.

2   If the operator of `upc_blocksizeof` has indefinite block size, the value of `upc_blocksizeof` is 0.

3   The type of the result is `size_t`.

Forward references: indefinite block size (6.4.2).

6.3.3   The `upc_elemsizeof` operator

>   upc_elemsizeof *unary-expression*
>
>   upc_elemsizeof (*type-name*)

Constraints

1   The `upc_elemsizeof` operator shall apply only to shared objects or shared-qualified types. All constraints on the `sizeof` operator [ISO/SEC00 Section 6.5.3.4] also apply to this operator.

Semantics

1   The `upc_elemsizeof` operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For non-array objects, `upc_elemsizeof` returns the same value as sizeof.

2   The type of the result is `size_t`.

6.3.4   Pointer-to-shared arithmetic

1   When an expression that has integer type is added to or subtracted from a pointer-to-shared, the result has the type of the pointer-to-shared operand. If the pointer-to-shared operand points to an element of a shared array object, and the shared array is large enough, the result points to an element of the shared array. If the shared array is declared with indefinite block size, the result of the pointer-to-shared arithmetic is identical to that described for normal C pointers in [ISO/SEC00 sec. 6.5.6], except that the thread of the new pointer shall be the same as that of the original pointer and the phase field is defined to always be zero. If the shared array has a definite block size, then the following example describes pointer arithmetic:

```
shared [B] int *p, *p1;  /* B a positive integer */
int i;

p1 = p + i;
```

2   After this assignment the following equations must hold in any UPC implementation. In each case the `div` operator indicates integer division rounding towards negative infinity and the `mod` operator returns the nonnegative remainder:[6]

```
upc_phaseof(p1) == (upc_phaseof(p) + i) mod B
upc_threadof(p1) == (upc_threadof(p)
                        + (upc_phaseof(p) + i) div B) mod THREADS
```

3   In addition, the correspondence between shared and private addresses and arithmetic is defined using the following constructs:

---

[6]The C "`%`" and "`/`" operators do not have the necessary properties

12

```
    T *P1, *P2;
    shared T *S1, *S2;

    P1 = (T*) S1;  /* legal if S1 has affinity to MYTHREAD */
    P2 = (T*) S2;  /* legal if S2 has affinity to MYTHREAD */
```

4   For all S1 and S2 that point to two distinct elements of the same shared array
    object which have affinity to the same thread:

- S1 and P1 shall point to the same object.

- S2 and P2 shall point to the same object.

- The expression ((`upc_addrfield` (S2) - `upc_addrfield`(S1)) shall eval-
  uate to the same value as ((P2 - P1) * sizeof(T)).

- If S1 < S2 then `upc_addrfield(S1)` shall be < `upc_addrfield(S2)`
  otherwise `upc_addrfield(S1)` shall be > `upc_addrfield(S2)`

Forward references: `upc_threadof` (7.3.5), `upc_phaseof` (7.3.6), `upc_addrfield`
(7.3.7).

6.3.5   Cast and Assignment Expressions

Constraints

1   A shared type qualifier shall not appear in a type cast of an object that is
    not shared-qualified, with the exception of the null pointer-to-shared.[7]

2   The cast of a pointer-to-shared to a pointer-to-private by a thread not having
    affinity with the referenced object has an undefined result.

Semantics

1   The casting or assignment from one pointer-to-shared to another in which
    either the type size or block size differs results in a pointer with a zero phase,
    unless one of the types is "shared void *", the generic pointer-to-shared.

2   If a generic pointer-to-shared is cast to a non-generic pointer-to-shared type
    with indefinite block size or with block size of one, the result is a pointer

---

[7]i.e., pointers-to-private cannot be cast to pointers-to-shared.

with a phase of zero. Otherwise, if the phase of the former pointer value is not within the range of possible phases of the latter pointer type, the result is undefined.

3   If a pointer-to-shared is cast[8] to a pointer-to-private[9] and the affinity of the shared data is not to the current thread, the result is undefined.

3   After the assignment

```
shared [B] T *s;

s = 0;
```

`s` is a null pointer-to-shared[10], and the operators `upc_threadof(s)` and `upc_phaseof(s)` evaluate to zero for all block sizes `B`.

6.3.6   Address Operators

Semantics

1   When the unary `&` is applied to a shared structure element of type `T`, the result has type `shared [] T *`.

6.4   Declarations

1   UPC extends the declaration ability of C to allow shared types, shared data layout across threads, and ordering constraint specifications.

Constraints

1   The declaration specifiers in a given declaration shall not include, either directly or through one or more typedefs, both **strict** and **relaxed**.

2   The declaration specifiers in a given declaration shall not specify more than one block size, either directly or indirectly through one or more typedefs.

Syntax

---

[8]As such pointers are not type compatible, explicit casts are required.

[9]References through such cast pointers behave exactly as if they were private accesses.

[10][ISO/SEC00] sec 6.3.2.3 and 6.5.16.1 imply that an implicit cast is allowed for zero and that all null pointers-to-shared compare equal.

14

1   The following is the declaration definition as per [ISO/SEC00] section 6.7, re-
    peated here for self-containment and clarity of the subsequent UPC extension
    specifications.

2   *declaration:*

> *declaration-specifiers init-declarator-list$_{opt}$* **;**

3   *declaration-specifiers:*

> *storage-class-specifier declaration-specifiers$_{opt}$*
>
> *type-specifier declaration-specifiers$_{opt}$*
>
> *type-qualifier declaration-specifiers$_{opt}$*
>
> *function-specifier declaration-specifiers$_{opt}$*

4   *init-declarator-list:*

> *init-declarator*
>
> *init-declarator-list* **,** *init-declarator*

5   *init-declarator:*

> *declarator*
>
> *declarator* **=** *initializer*

Forward references: strict and relaxed type qualifiers (6.4.2).

6.4.1   Type qualifiers

1   This subsection provides the UPC parallel extensions of section 6.7.3 in
    [ISO/SEC00].

Syntax

1   *type-qualifier:*

> **const**
>
> **restrict**
>
> **volatile**
>
> *shared-type-qualifier*
>
> *reference-type-qualifier*

6.4.2 The shared and reference type qualifiers

Syntax

1 *shared-type-qualifier:*

> **shared** *layout-qualifier$_{opt}$*

2 *reference-type-qualifier:*

> **relaxed**

> **strict**

3 *layout-qualifier:*

> [*constant-expression$_{opt}$*]

> [*]

Constraints

1 A reference type qualifier shall appear in a qualifier list only when the list also contains a shared type qualifier.

2 A shared type qualifier can appear anywhere a type qualifier can appear except that it shall not appear in the specifier qualifier list of a structure declaration unless it qualifies a pointer type.

3 A layout qualifier of [*] shall not appear in the declaration specifiers of a pointer.

4 A layout qualifier shall not appear in the type qualifiers for a pointer to void type.

Semantics

1 An object that has shared-qualified type shall exist in shared memory space and not in private space. Any thread may reference a shared object. Shared objects are placed in memory based on an affinity to a particular thread.

2 References to shared objects, either directly or via pointer-to-shared indirection, shall be either strict or relaxed. Strict and relaxed references behave as described in section 5.1.2.3 of this document.

3 A reference shall be determined to be strict or relaxed as follows. If the referenced type is strict-qualified or relaxed-qualified, the reference shall be strict or relaxed, respectively. Otherwise the reference shall be determined

to be strict or relaxed by the UPC pragma rules, as described in section 6.6.1 of this document.

4   The layout qualifier dictates the blocking factor for the type being shared qualified. This factor is the nonnegative number of consecutive elements (when evaluating pointer-to-shared arithmetic and array declarations) which have affinity to the same thread. If the optional constant expression is 0 or is not specified, all objects have affinity to the same thread. If there is no layout qualifier, the blocking factor has the default value (1). The blocking factor is also referred to as the block size.

5   A layout qualifier indicating that all array elements have affinity to the same thread is said to specify indefinite block size.

6   The block size is a part of the type compatibility[11]

7   A shared void* pointer is assignment compatible with any pointer-to-shared type.

8   If the layout qualifier is of the form '[*]', the shared object is distributed as if it had a block size of

```
( sizeof(a) / upc_elemsizeof(a) + THREADS - 1 ) / THREADS,
```

where 'a' is the array being distributed.

9   EXAMPLE 1: declaration of a shared scalar

```
strict shared int y;
```

`strict shared` is the type qualifier.

10 EXAMPLE 2: automatic storage duration

```
void foo (void) {
shared int x;  /* a shared automatic variable is not allowed  */
shared int* y; /* a pointer to shared is allowed  */
int * shared z; /*a shared automatic variable is not allowed*/
... }
```

---

[11]This is a powerful statement which allows, for example, that in an implementation `sizeof(shared int *)` may differ from `sizeof (shared [10] int *)` and if T and S are pointer-to-shared types with different block sizes, then T* and S* cannot be aliases.

11 EXAMPLE 3: inside a structure

```
struct foo {
shared int x;  /* this is not allowed  */
shared int* y; /* a pointer to a shared object is allowed  */
};
```

Forward references: shared array (6.4.3.2), pointer declarator (6.4.3.1).

6.4.3   Declarators

Syntax

1   The following is the declarator definition as per [ISO/SEC00] section 6.7.5, repeated here for self-containment and clarity of the subsequent UPC extension specifications.

2   *declarator:*

      pointer$_{opt}$ direct-declarator

3   *direct-declarator:*

      identifier

      (declarator)

      direct-declarator [*type-qualifier-list$_{opt}$ assignment-expression$_{opt}$*]

      direct-declarator [**static** *type-qualifier-list$_{opt}$ assignment-expression*]

      direct-declarator [*type-qualifier-list* **static** *assignment-expression*]

      direct-declarator [*type-qualifier-list$_{opt}$* *****]

      direct-declarator (*parameter-type-list*)

      direct-declarator (*identifier-list$_{opt}$*)

4   *pointer:*

      * *type-qualifier-list$_{opt}$*

      * *type-qualifier-list$_{opt}$ pointer*

5   *type-qualifier-list:*

      type-qualifier

type-qualifier-list type-qualifier

Constraints

1 No type qualifier list shall specify more than one block size, either directly or indirectly through one or more typedefs.[12]

2 No type qualifier list shall include both `strict` and `relaxed` either directly or indirectly through one or more typedefs.

3 `shared` shall not appear in a declarator which has automatic storage duration, unless it qualifies a pointer type.

Semantics

1 All static non-array shared-qualified objects have affinity with thread zero.

2 Only pointer type members of a structure or union may be shared-qualified.[13]

6.4.3.1 Pointer declarators

1 This subsection provides the UPC parallel extensions of section 6.7.5.1 in [ISO/SEC00].

Semantics

1 A shared reference which is cast to non-shared will lose all qualities pertaining to being shared.

2 Shared objects with affinity to a given thread can be accessed by either pointers-to-shared or pointers-to-private of that thread.

3 EXAMPLE 1:

```
int i, *p;
shared int *q;
q = (shared int *)p;        /* is not allowed */
if (upc_threadof(q) == MYTHREAD)
    p = (int *) q;          /* is allowed */
```

---

[12]While layout qualifiers are most often seen in array or pointer declarators, they are legal in all declarators. For example, `shared [3] int y` is a legal declarator.

[13]E.g., `struct S1 { shared char * p1; };` is legal, while `struct S2 { char * shared p2; };` is not.

6.4.3.2 Array declarators

1   This subsection provides the UPC parallel extensions of section 6.7.5.2 in [ISO/SEC00].

Constraints

1   When a UPC program is translated in the "dynamic THREADS" environment and the type of the array is shared-qualified but not indefinite layout-qualified, the THREADS lvalue shall occur exactly once in one dimension of the array declarator (including through typedefs). Further, in such cases, the THREADS lvalue shall only occur either alone or when multiplied by a constant expression.

Semantics

1   Elements of shared arrays are distributed in a round robin fashion, by chunks of block-size elements, such that the i-th element has affinity with thread (floor (i/block_size) `mod THREADS`).

2   In an array declaration, the type qualifier applies to the elements.

3   For any shared array, `a`, `upc_phaseof (&a)` is zero.

4   EXAMPLE 1: declarations legal in either static or dynamic translation environments:

```
shared int x [10*THREADS];
shared [] int x [10];
```

5   EXAMPLE 2: declarations legal only in static translation environment:

```
shared int x [10+THREADS];
shared [] int x [THREADS];
shared int x [10];
```

6   EXAMPLE 3: declaration of a shared array

```
shared [3] int x [10];
```

`shared [3]` is the type qualifier of an array, `x`, of 10 integers. `[3]` is the layout qualifier.

7   EXAMPLE 4:

```
typedef int S[10];
shared [3] S T[3*THREADS];
```

`shared [3]` applies to the underlying type of T, which is int, regardless of the typedef. The array is blocked as if it were declared:

```
shared [3] int T[3*THREADS][10];
```

8   EXAMPLE 5:

```
shared [] double D[100];
```

All elements of the array D have affinity to thread 0. No `THREADS` dimension is allowed in the declaration of D.

```
shared [] long *p;
x = p[i];
```

All elements referenced by subscripting or otherwise dereferencing p have affinity to the same thread. That thread may be any thread; it does not have to be thread 0.

6.5   Statements and blocks

1   This subsection provides the UPC parallel extensions of section 6.8 in [ISO/SEC00].

Syntax

1   *statement:*

labeled-statement

compound-statement

expression-statement

> *selection-statement*
>
> *iteration-statement*
>
> *jump-statement*
>
> *synchronization-statement*

6.5.1 Barrier Statements

Syntax

1 *synchronization-statement:*

> **upc_notify** *expression$_{opt}$* ;
>
> **upc_wait** *expression$_{opt}$* ;
>
> **upc_barrier** *expression$_{opt}$* ;
>
> **upc_fence** ;

Constraints

1 *expression* shall be an *integer expression.*

2 Each thread shall execute an alternating sequence of `upc_notify` and `upc_wait` statements, starting with a `upc_notify` and ending with a `upc_wait` statement. A synchronization phase consists of the execution of all statements between the completion of one `upc_wait` and the start of the next.

Semantics

1 A `upc_wait` statement completes, and the thread enters the next synchronization phase, only after all threads have completed the `upc_notify` statement in the current synchronization phase.[14] `upc_wait` and `upc_notify` are *collective* operations.

2 The `upc_fence` statement is equivalent to a *null* strict reference. This insures that all shared references issued before the fence are complete before any after it are issued.[15]

---

[14]Therefore, all threads are entering the same synchronization phase as they complete the `upc_wait` statement.

[15]One implementation of `upc_fence` may be achieved by a null strict reference: { `static shared strict int x; x = x;`}

4   A null strict reference is implied before a `upc_notify` statement and after a `upc_wait` statement.[16]

5   The `upc_wait` statement will generate a runtime error if the value of its expression does not equal the value of the expression by the `upc_notify` statement for the current synchronization phase. No error will be generated if either statement does not have an expression.

6   The `upc_wait` statement will generate a runtime error if the value of its expression differs from any expression on the `upc_wait` and `upc_notify` statements issued by any thread in the current synchronization phase. No error will be generated from a "difference" involving a statement for which no expression is given.

7   The `upc_barrier` statement is equivalent to the compound statement[17]:

```
{ upc_notify barrier_value; upc_wait barrier_value; }
```

8   The barrier operations at thread startup and termination have a value of *expression* which is not in the range of user expressible values.

9   EXAMPLE 1: The following will result in a runtime error:

```
{ upc_notify; upc_barrier; upc_wait; }
```

as it is equivalent to

```
{ upc_notify; upc_notify; upc_wait; upc_wait; }
```

6.5.2   Iteration statements

1   This subsection provides the UPC parallel extensions of section 6.8.5 in [ISO/SEC00].

Syntax

---

[16]This implies that shared references executed after the `upc_notify` and before the `upc_wait` may occur in either the synchronization phase containing the `upc_notify` or the next on different threads.

[17]This equivalence is explicit with respect to matching expressions in semantic 6 and collective status in semantic 1.

1   *iteration-statement:*

        while (*expression*) *statement*

        do *statement* **while** (*expression*) ;

        for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

        for (*declaration-expression*$_{opt}$; *expression*$_{opt}$) *statement*

        upc_forall (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$; *affinity*$_{opt}$)
statement

2   *affinity:*

        expression$_{opt}$

        continue

Constraints:

1   The *expression* for affinity shall have pointer-to-shared type or integer type.

Semantics:

1   `upc_forall` is a *collective* operation in which, for each execution of the loop body, the controlling expression and affinity expression are *single-valued*.[18]

2   The *affinity* field specifies the executions of the loop body which are to be performed by a thread.

3   When *affinity* is of pointer-to-shared type, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value of `upc_threadof`(*affinity*). Each iteration of the loop body is executed by precisely one thread.

4   When *affinity* is an integer expression, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value *affinity* `mod` `THREADS`.

5   When *affinity* is `continue` or not specified, each loop body of the `upc_forall` statement is performed by every thread and semantic 1 does not apply.

6   If the loop body of a `upc_forall` statement contains one or more `upc_forall` statements, either directly or through one or more function calls, the construct is called a "nested upc_forall" statement. In a "nested upc_forall", the

---

[18]Note that single-valued implies that all thread agree on the total number of iterations, their sequence, and which threads execute each iteration.

outermost `upc_forall` statement that has an *affinity* expression which is not `continue` is called the "controlling upc_forall" statement. All `upc_forall` statements which are not "controlling" in a "nested upc_forall" behave as if their *affinity* expressions were `continue`.

7 If the execution of any loop body of a `upc_forall` statement produces a side-effect which affects the execution of another loop body of the same `upc_forall` statement which is executed by a different thread[19], the behavior is undefined.

8 If any thread terminates or executes a `upc_barrier`, `upc_notify`, or `upc_wait` statement within the dynamic scope of a `upc_forall` statement, the result is undefined. If any thread terminates a `upc_forall` statement using a `break`, `goto` , or `return` statement, the result is undefined. If any thread enters the body of a `upc_forall` statement using a `goto` statement, the result is undefined.[20]

9 EXAMPLE 1: Nested `upc_forall`:

```
main () {
    int i,j,k;
    shared float *a, *b, *c;

    upc_forall(i=0; i<N; i++; continue)
        upc_forall(j=0; j<N; j++; &a[j])
            upc_forall (k=0; k<N; k++; &b[k])
                a[j] = b[k] * c[i];
}
```

This example executes all iterations of the "i" and "k" loops on every thread, and executes iterations of the "j" loop on those threads where `upc_threadof` (`&a[j]`) equals the value of `MYTHREAD`.

---

[19]This semantic implies that side effects on the same thread have defined behavior, just like in the `for` statement.

[20]The `continue` statement behaves as defined in [ISO/SEC 00; Section 6.8.6.2].; equivalent to a `goto` the end of the loop body.

6.6 Preprocessing directives

1   This subsection provides the UPC parallel extensions of section 6.10 in [ISO/SEC00].

6.6.1   UPC pragmas

Semantics

1   If the preprocessing token `upc` immediately follows the `pragma`, then no macro replacement is performed and the directive shall have one of the following forms:

```
#pragma upc strict

#pragma upc relaxed
```

2   These pragmas affect the strict or relaxed categorization of references to shared objects where the referenced type is neither strict-qualified nor relaxed-qualified. Such references shall be strict if a strict pragma is in effect, or relaxed if a relaxed pragma is in effect.

3   Shared references which are not categorized by either referenced type or by these pragmas behave in an implementation defined manner in which either all such references are strict or all are relaxed. Users wishing portable programs are strongly encouraged to categorize all shared references either by using type qualifiers, these directives, or by including upc_strict.h or upc_relaxed.h.

4   The pragmas shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When they are outside external declarations, they apply until another such pragma or the end of the translation unit. When inside a compound statement, they apply until the end of the compound statement; at the end of the compound statement the state of the pragmas is restored to that preceding the compound statement. If these pragmas are used in any other context, their behavior is undefined.

6.6.2   Predefined macro names

1   The following macro name shall be defined by the implementation:

__UPC__ The integer constant 1, indicating a conforming implementation.

2   The following macro names are conditionally defined by the implementation:

__UPC_DYNAMIC_THREADS__ The integer constant 1 in the dynamic THREADS translation environment.

__UPC_STATIC_THREADS__ The integer constant 1 in the static THREADS translation environment.

7   Library

7.1   Standard headers

1   This subsection provides the UPC parallel extensions of section 7.1.2 in [ISO/SEC00].

2   The standard headers are

```
<upc_strict.h>
<upc_relaxed.h>
<upc.h>
```

3   upc_strict.h shall contain at least:

```
#pragma upc strict
#include <upc.h>
```

4   upc_relaxed.h shall contain at least:

```
#pragma upc relaxed
#include <upc.h>
```

7.2 UPC utilities <upc.h>

1   This subsection provides the UPC parallel extensions of section 7.20 in [ISO/SEC00]. All of the characteristics of library functions described in section 7.1.4 in [ISO/SEC00] apply to these as well.

7.2.1 Termination of all threads

Synopsis

```
upc_global_exit(int status)
```

Description

1   upc_global_exit() flushes all I/O, releases all memory, and terminates the execution for all active threads.

7.2.2 Shared memory allocation functions

1   The UPC memory allocation functions return, if successful, a pointer-to-shared which is suitably aligned so that it may be assigned to a pointer-to-shared of any type. The pointer has zero phase and points to the start of the allocated space. If the space cannot be allocated, a null pointer-to-shared is returned.

7.2.2.1 The upc_global_alloc function

Synopsis

1
```
#include <upc.h>

shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
 nblocks : number of blocks
 nbytes : block size
```

Description

1   The upc_global_alloc allocates shared space compatible with the declaration:

```
                     shared [nbytes] char[nblocks * nbytes].
```

2   The `upc_global_alloc` function is not a *collective* function. If called by multiple threads, all threads which make the call get different allocations.

7.2.2.2   The `upc_all_alloc` function

Synopsis

1
```
    #include <upc.h>

    shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
      nblocks : number of blocks
      nbytes : block size
```

Description

1   `upc_all_alloc` is a *collective* function with *single-valued* arguments.

2   The `upc_all_alloc` function allocates shared space compatible with the following declaration:

```
    shared [nbytes] char[nblocks * nbytes].
```

3   The `upc_all_alloc` function returns the same pointer value on all threads.

4   The dynamic lifetime of an allocated object extends from the time any thread completes the call to `upc_all_alloc` until any thread has deallocated the object.

7.2.2.3   The `upc_alloc` function

Synopsis

1
```
    #include <upc.h>

    shared void *upc_alloc(size_t nbytes);
     nbytes : total number of bytes to allocate
```

Description

1   The upc_alloc function allocates shared space of at least **nbytes** bytes with affinity to the calling thread.

2   upc_alloc is similar to malloc() except that it returns a pointer-to-shared value. It is not a *collective* function.

7.2.2.4   The upc_local_alloc function *deprecated*

Synopsis

1
```
#include <upc.h>

shared void *upc_local_alloc(size_t nblocks, size_t nbytes);
 nblocks : number of blocks
 nbytes : block size
```

Description

1   The upc_local_alloc function is deprecated and should not be used. UPC programs should use the upc_alloc function instead. Support may be removed in future versions of this specification.

2   The upc_local_alloc function allocates shared space of at least **nblocks * nbytes** bytes with affinity to the calling thread.

3   upc_local_alloc is similar to malloc() except that it returns a pointer-to-shared value. It is not a *collective* function.

7.2.2.5   The upc_free function

Synopsis

1
```
#include <upc.h>

void upc_free(shared void *ptr);
```

Description

1   The `upc_free` function frees the dynamically allocated shared memory pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `upc_alloc`, `upc_global_alloc`, `upc_all_alloc`, or `upc_local_alloc`, function, or if the space has been deallocated by a previous call, by any thread,[21] to `upc_free`, the behavior is undefined.

7.2.3   Pointer-to-shared manipulation functions

7.2.3.1   The `upc_threadof` function

Synopsis

1
```
    #include <upc.h>

    size_t upc_threadof(shared void *ptr);
```

Description

1   The `upc_threadof` function returns the number of the thread that has affinity to the shared object pointed to by `ptr`.

7.2.3.2   The `upc_phaseof` function

Synopsis

1
```
    #include <upc.h>

    size_t upc_phaseof(shared void *ptr);
```

Description

1   The `upc_phaseof` function returns the phase field of the pointer-to-shared argument.

---

[21]i.e., only one thread may call `upc_free` for each allocation

7.2.3.3 The upc_resetphase function

Synopsis

```
1       #include <upc.h>

        shared void *upc_resetphase(shared void *ptr);
```

Description

1   The upc_resetphase function returns a pointer-to-shared which is identical
to its input except that it has zero phase.

7.2.3.4 The upc_addrfield function

Synopsis

```
1       #include <upc.h>

        size_t upc_addrfield(shared void *ptr);
```

Description

1   The upc_addrfield function returns an implementation-defined value re-
flecting the "local address" of the object pointed to by the pointer-to-shared
argument.

7.2.3.5 The upc_affinitysize function

Synopsis

```
1       #include <upc.h>

        size_t upc_affinitysize(size_t nblocks, size_t nbytes,
                                size_t threadid);
          nblocks: the number of blocks in the allocation
          nbytes: the number of bytes in a block
          threadid: the thread whose affinitysize is to be evaluated
```

Description

1   `upc_affinitysize` is a convenience function which calculates the exact size of the local portion of the data in a shared object with affinity to a given thread.

2   In the case of a dynamically allocated shared object, the `nblocks` and `nbytes` arguments shall be exactly those passed to `upc_global_alloc` or `upc_all_alloc` when the object was allocated.

3   In the case of a statically allocated shared object with declaration:

```
shared [b] t d[s];
```

the `nbytes` argument shall be `b * sizeof (t)` and the `nblocks` argument shall be `sizeof (d) / (b * sizeof (t)))`. If block size is unspecified, it shall be 1.

4   `threadid` shall be a value in 0..(`THREADS`-1).

7.2.4 Lock functions

7.2.4.1 Type

1   The type declared is

```
upc_lock_t
```

2   The type `upc_lock_t` is an opaque UPC type. `upc_lock_t` is a shared datatype with incomplete type (as defined in section 6.2.5 of [ISO/SEC00]). Objects of type `upc_lock_t` may therefore only be manipulated through pointers.

7.2.4.2 The `upc_global_lock_alloc` function

Synopsis

1
```
#include <upc.h>

upc_lock_t *upc_global_lock_alloc(void);
```

Description

1　The `upc_global_lock_alloc` function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.

2　The `upc_global_lock_alloc` function is not a *collective* function. If called by multiple threads, all threads which make the call get different allocations.

7.2.4.3　The `upc_all_lock_alloc` function

Synopsis

1　　　`#include <upc.h>`

　　　`upc_lock_t *upc_all_lock_alloc(void);`

Description

1　The `upc_all_lock_alloc` function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.

2　The `upc_all_lock_alloc` is a *collective* function. The return value on every thread points to the same lock object.

7.2.4.4　The `upc_lock_free` function

Synopsis

1　　　`#include <upc.h>`

　　　`void upc_lock_free(upc_lock_t *ptr);`

Description

1　The `upc_lock_free` function frees all resources associated with the dynamically allocated upc_lock_t pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `upc_global_lock_alloc` or `upc_all_lock_alloc` function, or if the lock has been deallocated by a previous call to `upc_lock_free,` the behavior is undefined.

2   `upc_lock_free` succeeds regardless of whether the referenced lock is currently unlocked or currently locked (by any thread).

3   Any subsequent calls to locking functions from any threads using `ptr` have undefined effects.

7.2.4.5 The `upc_lock` function

Synopsis

1   ```
        #include <upc.h>

        void upc_lock(upc_lock_t *ptr);
        ```

**Description**

1    The `upc_lock` function locks a shared variable, of type `upc_lock_t,` pointed to by the pointer given as argument.

2    If the lock is not used by another thread, then the thread making the call gets the lock and the function returns. Otherwise, the function keeps trying to get access to the lock.

3    A null strict reference is implied after a call to `upc_lock()`.

4    If the calling thread is already holding the lock referenced by `ptr` (i.e., it has previously locked it using `upc_lock()` or `upc_lock_attempt()`, but not unlocked it), the result is undefined.


### 7.2.4.6  The `upc_lock_attempt` function

**Synopsis**

1       ```
        #include <upc.h>

        int upc_lock_attempt(upc_lock_t *ptr);
        ```

Description

1   The `upc_lock_attempt` function tries to lock a shared variable, of type `upc_lock_t`, pointed to by the pointer given as argument.

2 If the lock is not used by another thread, then the thread making the call gets the lock and the function returns 1. Otherwise, the function returns 0.

3 A null strict reference is implied after a call to `upc_lock_attempt()` that returns 1.

4 If the calling thread is already holding the lock referenced by `ptr` (i.e., it has previously locked it using `upc_lock()` or `upc_lock_attempt()`, but not unlocked it), the result is undefined.

7.2.4.7 The `upc_unlock` function

Synopsis

1
```
    #include <upc.h>

    void upc_unlock(upc_lock_t *ptr);
```

Description

1 The `upc_unlock` function frees the lock and does not return any value.

2 A null strict reference is implied before a call to `upc_unlock()`.

7.2.5 Shared string handling functions

7.2.5.1 The `upc_memcpy` function

Synopsis

1
```
    #include <upc.h>

    void upc_memcpy(shared void *dst, shared const void *src,
                    size_t n);
```

Description

1 The `upc_memcpy` function copies a block of memory from one shared memory area to another shared memory area. The number of bytes copied is n. If copying takes place between objects that overlap, the behavior is undefined.

2 The `upc_memcpy` function treats the `dst` and `src` pointers as if each of them pointed to a shared memory space on a single thread and therefore had type:

```
shared [] char[n]
```

The effect is equivalent to copying the entire contents from one shared array
with this type (the `src` array) to another shared array with this type (the
`dst` array).

7.2.5.2   The upc_memget function

Synopsis

1   `#include <upc.h>`

      `void upc_memget(void *dst, shared const void *src, size_t n);`

Description

1   The `upc_memget` function copies a block of memory from a shared memory
area to a private memory area on the calling thread. The number of bytes
copied is n. If copying takes place between objects that overlap, the behavior
is undefined.

2   The `upc_memget` function treats the `src` pointer as if it pointed to a shared
memory space on a single thread and therefore had type:

```
shared [] char[n]
```

The effect is equivalent to copying the entire contents from one shared array
with this type (the `src` array) to a local array (the `dst` array) declared with
the type

```
char[n]
```

7.2.5.3   The upc_memput function

Synopsis

1   `#include <upc.h>`

      `void upc_memput(shared void *dst, const void *src, size_t n);`

Description

1   The `upc_memput` function copies a block of memory from the calling thread's private memory area to a shared memory area. The number of bytes copied is n. If copying takes place between objects that overlap, the behavior is undefined.

2   The `upc_memput` function is equivalent to copying the entire contents from a local array (the `src` array) declared with the type

```
char[n]
```

to a shared array (the `dst` array) with the type

```
shared [] char[n]
```

7.2.5.4   The `upc_memset` function

Synopsis

1   `#include <upc.h>`

        `void upc_memset(shared void *dst, int c, size_t n);`

Description

1   The `upc_memset` function copies the value of `c`, converted to an `unsigned char`, to a shared memory area. The number of bytes set is `n`.

2   The `upc_memset` function treats the `dst` pointer as if it pointed to a shared memory space on a single thread and therefore had type:

```
shared [] char[n]
```

The effect is equivalent to setting the entire contents of a shared array with this type (the `dst` array) to the value `c`.

References

[CARLSON99] W. W. Carlson, J. M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. CCS-TR-99-157. IDA/CCS, Bowie, Maryland. May, 1999.

[ISO/SEC00] ANSI. Programming Languages-C. ISO/SEC 9899. May, 2000.

A UPC versus ANSI C section numbering

UPC specifications subsection	ANSI C specifications subsection
1	1
2	2
3	3
4	4
5	5
6	6
6.1	6.1
6.2	6.4.2.2
6.3	6.5
6.3.6	6.5.3.1
6.4	6.7
6.4.1	6.7.3
6.4.3	6.7.5
6.5	6.8
6.6	6.10
7	7
7.1	7.1.2

Table A1. Mapping UPC subsections to ANSI C specifications subsections